ArchExplorer: Microarchitecture Exploration Via Bottleneck Analysis

Chen Bai*
The Chinese University of Hong Kong
DAMO Academy, Alibaba Group
cbai@cse.cuhk.edu.hk

Jiayi Huang†
The Hong Kong University of Science and Technology (Guangzhou)
hjy@hkust-gz.edu.cn

Xuechao Wei
DAMO Academy, Alibaba Group
xuechao.wxc@alibaba-inc.com

Yuzhe Ma
The Hong Kong University of Science and Technology (Guangzhou)
yuzhema@hkust-gz.edu.cn

Sicheng Li
DAMO Academy, Alibaba Group
sicheng.li@alibaba-inc.com

Hongzhong Zheng
DAMO Academy, Alibaba Group
hongzhong.zheng@alibaba-inc.com

Bei Yu
The Chinese University of Hong Kong
byu@cse.cuhk.edu.hk

Yuan Xie
The Hong Kong University of Science and Technology
DAMO Academy, Alibaba Group
y.xie@alibaba-inc.com

ABSTRACT

Design space exploration (DSE) for microarchitecture parameters is an essential stage in microprocessor design to explore the trade-offs among performance, power, and area (PPA). Prior work either employs excessive expert efforts to guide microarchitecture parameter tuning or demands high computing resources to prepare datasets and train black-box prediction models for DSE.

In this work, we aim to circumvent the domain knowledge requirements through automated bottleneck analysis and propose ArchExplorer, which reveals microarchitecture bottlenecks to guide DSE with much fewer simulations. ArchExplorer consists of a new graph formulation of microexecution, an optimal critical path construction algorithm, and hardware resource reassignment strategies. Specifically, the critical path is constructed from the microexecution to uncover the performance-critical microarchitecture bottlenecks, which facilitates ArchExplorer to reclaim the hardware budgets of performance-insensitive structures that consume unnecessary power and area. These budgets are then reassigned to the microarchitecture bottlenecks for performance boost while maintaining the power and area constraints under the total budget envelope. Experiments show that ArchExplorer can find better PPA Pareto-optimal designs, achieving an average of 6.80% higher Pareto hypervolume using at most 74.63% fewer simulations compared to the state-of-the-art approaches.

*This work is done during Chen Bai's internship at Alibaba DAMO Academy.
†Part of the work was done while the author was with Alibaba DAMO Academy.

MICRO '23, October 28 – November 1, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614289

CCS CONCEPTS

• **Computer systems organization** → **Superscalar architectures**.

KEYWORDS

Microprocessor, Microarchitecture, Design Space Exploration

ACM Reference Format:
Chen Bai, Jiayi Huang, Xuechao Wei, Yuzhe Ma, Sicheng Li, Hongzhong Zheng, Bei Yu, and Yuan Xie. 2023. ArchExplorer: Microarchitecture Exploration Via Bottleneck Analysis . In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3614289

1 INTRODUCTION

The microprocessor design cycle at the concept phase (logic level details are unavailable) requires abundant performance-power-area (PPA) trade-off evaluations. Architects rely on practical algorithms to explore optimal architecture parameters, achieving sweet PPA balance within a limited time budget. For example, five of Tenstorrent's competitive RISC-V microprocessor implementations are fast prototyped based on one design within a year to face diverse PPA design targets [5].

Microarchitecture exploration is a design space exploration (DSE) problem aiming to find performance-power-area Pareto-optimal microprocessor parameters. Architects are often confronted with billions or trillions of parameter combinations. And one combination takes a high runtime to acquire the PPA results via detailed simulations. The problem is not new, and the industry and academia have proposed many solutions.

In industry, microarchitecture parameters are determined upon extensive simulation results analysis and the expertise of architects. However, the fact that architects' domain knowledge can fall into a personal bias is a concern. The question remains whether the solution given by architects is optimal or how many benefits we can

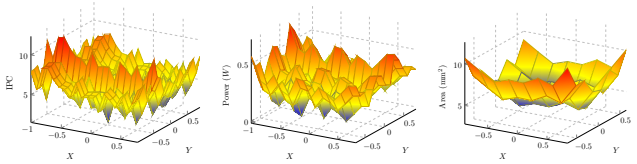
gain based on a sub-optimal solution. In academia, researchers adopt mechanistic models or black-box methodologies. Mechanistic models investigate the relations between PPA values and microarchitecture parameters by unfolding the microexecutions. Interpretable equations are constructed for a single component [26, 27, 48, 55] or the entire microprocessor [19, 31, 43]. Microarchitecture exploration is conducted by sweeping the design space or fast evaluation with the mechanistic model. Nevertheless, the model requires immense domain knowledge to build and verify. More commonly-applied solutions use black-box methodologies [7, 8, 10, 12, 15, 16, 28, 32, 34, 35, 37, 60]. The methods train a black-box model with machine-learning techniques via a large data set. Researchers prefer them since better results are often achieved [8, 12, 28, 35, 37]. However, black-box methods are not a silver bullet. They require high computing resources to construct the data set for training [12, 28]. Another criticism of the black-box method is that blindly (purely driven by the algorithm rather than tightly coupled with expertise) exploring microarchitectures seems naive since architects already know the characteristics of most designs.

Goal and Approach: We aim to solve the problem by evading the limitations of current mainstream methodologies. Specifically, we circumvent the massive domain knowledge access required by building mechanistic models and mitigate the high computing demands of black-box methods. The key to achieving the goal is *DSE via automated bottleneck analysis*, where the bottlenecks are the factors that hinder the program execution progress. We reduce the demand for expert knowledge via automated bottleneck detection and elimination. Meanwhile, our method asks for fewer computing demands compared to black-box methodologies.

Rationales: Assume a perfect machine has unlimited hardware resources. The factors constraining the perfect machine's performance are only the program's true data dependencies (*viz.*, read-after-write dependencies). While in a real machine, architecture parameters like the instruction queue entries set the resource constraints. Due to the constraints, contentions for limited resources exist in the microexecution. And the contention produces two distinct types of resources: 1) deficient and exhausted and 2) abundant and idle. Hence, a real machine leads to unbalanced usage of resources. The usage dependencies of deficient resources are another factor that blocks instructions from progressing. A *balanced microarchitecture* can simultaneously maximize the utilization of each hardware resource. In other words, the microarchitecture makes the best use of every resource. We refer to a bottleneck as insufficient hardware resource, which is exhausted by instructions and results in high program runtime. Accurate and efficient identification of types of hardware resources is the first principle to find a balanced microarchitecture.

Findings and Design Principles: Jouppi decoupled the microprocessor performance into benchmark and machine parallelism [29]. Our observation is that the relations between resource constraints and machine parallelism are similar to the cask effect [1]. Namely,

[1]The cask effect is a terminology from the Peter principle [49]. It states that in a hierarchy, every employee tends to rise to its level of incompetence. We adopt the term to broadly summarize the insight of our approach.



(a) Performance (b) Power (c) Area

Figure 1: A visualization of the design space for 458.sjeng. Each microarchitecture is reduced to two dimensions through t-SNE [57] to facilitate the visualization of PPA distributions.

the most insufficient resource largely determines machine parallelism. For example, assigning more to such resources can achieve 23.05% performance improvement, and the PPA trade-off becomes 27.42% better. To identify the type of resource, two requirements should be satisfied. The utilization status of each resource in the microexecution should be captured. And whether the overlapping events matter for the execution time should be considered. The latter requirements call for a global view of the entire microexecution, which the critical path can represent [22]. We summarize two design principles for DSE via bottleneck analysis. *First, the dependencies contributing to execution time should be captured as much as possible.* Approximating the resource utilization more accurately benefits the DSE. *Second, concurrent events should be distinguishable.* The distinguishability is essential in accurately estimating bottleneck contributions to the execution time. Accordingly, we propose a new graph model formulation based on critical path analysis [24, 36, 44, 45, 56] to implement the two design principles.

Contributions:

1) The design principles and a new graph model formulation to characterize the microexecution.

2) The induced graph model and an optimal critical path construction algorithm based on dynamic programming to investigate overlapped events.

3) Evaluations show that our DSE method ArchExplorer can find better Pareto PPA microarchitectures with an average of 6.80% higher Pareto hypervolume using at most 74.63% fewer simulations compared to state-of-the-art approaches.

Paper Organization: The rest of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 states lessons and design principles. Section 4 provides the ArchExplorer approach. Section 5 and Section 6 are for experiments. Section 7 presents some discussions. Section 8 supplements additional related works and Section 9 concludes the paper.

2 BACKGROUND & MOTIVATION

2.1 Challenges in Microarchitecture DSE

The two challenges of the problem are the extremely large and non-smooth design space and high simulation overheads to get the performance values.

The design space is complicated due to the non-linear relations between parameters and performance and power values [10, 30].

Table 1: A baseline microarchitecture specification

Components	Hardware Resources
Pipeline width	4
Fetch buffer size in bytes	64
Fetch queue size in μ-ops	32
Branch predictor unit	local/global/choice predictor of the tournament: 2048/8192/8192 RAS: 16, BTB: 4096
ROB/IQ/LQ/SQ	50/32/24/24
Physical register	Int RF: 50, Floating-point RF: 50
Functional unit	IntALU: 3, IntMultDiv: 1, FpALU: 2 FpMultDiv: 1, RdWrPort: 1
L1 I$	2-way, 32 KB, 2 cycles
L1 D$	2-way, 32 KB, 2 cycles
IPC/Power/Area	0.9418/0.2027 W/5.6609 mm^2

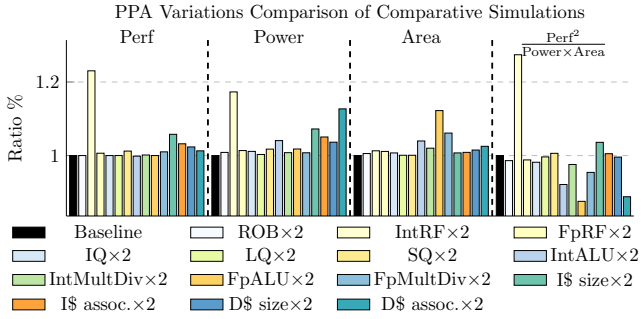


Figure 2: Each bar represents the microarchitecture's metric in %. The bar, *e.g.*, "ROB ×2", indicates the microarchitecture is the same as the baseline except that it doubles ROB. Perf^2/(Power × Area) denotes the PPA trade-off.

To visualize the design space intuitively, we use GEM5's [9, 39] out-of-order RISC-V model [33]. Each sampled design is extensively evaluated with the SPEC CPU2006 benchmark suite [25], and power and area values are reported from McPAT [38]. Figure 1 reveals the design space *w.r.t.* PPA values for 458.sjeng. Figure 1(a) and Figure 1(b) show many extrema in the space, and the changes between these extrema are non-smooth, indicating that the performance and power design space is complicated. The area design space, as referred to in Figure 1(c), is relatively flat since linear relations exist between parameters and area.

Obtaining the performance value for a design requires high-fidelity simulation with representative benchmarks, which can result in high runtime costs even with a fast simulator [11].

2.2 Bottleneck Analysis Matters in DSE

Although the design space is complicated, improving machine parallelism by removing microarchitecture bottlenecks can significantly enhance the PPA trade-off, as demonstrated by an example of comparative simulations. Table 1 lists a baseline microarchitecture, and we evaluate it with SPEC CPU2017 Simpoints [47]. The average performance and power values of all workloads are reported. Comparative simulations are conducted for each new microarchitecture by doubling individual parameters, as shown in Figure 2. Firstly, doubling parameters like the number of floating-point arithmetic

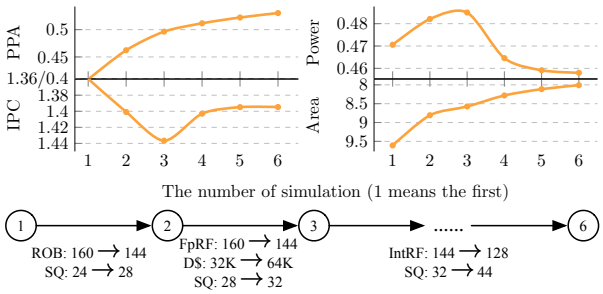


Figure 3: Search by following a series of small changes stepwise. PPA denotes Perf^2/(Power × Area).

logic units (FpALU) worsens power and area without improving performance, indicating redundant resources waste the design budget. Secondly, doubling the number of physical integer registers (IntRF) improves performance by 23.05% and enhances the PPA trade-off by 27.42%. We investigate the simulation trace to find out the root cause. We find that most instructions are stalled in the rename stage due to insufficient physical integer registers. For instance, this bottleneck results in 25.71% of instructions in 657.xz_s and 18.94% for 625.x264_s getting stalled during renaming.

We apply a straightforward heuristic to find a balanced microarchitecture: *assigning necessary hardware resources and reducing redundant ones.* We adjust resources based on the degree of necessity, which is the ratio of delayed instructions due to the resource's insufficiency. If the necessity is top-ranked, we increase the resource, and if it is zero, we decrease it. Figure 3 shows a stepwise search by reusing the design space (Table 4), where we analyze the simulation trace manually to calculate the necessity for each resource. With only six simulations, we improve performance by 2.59%, reduce power and area by 2.66% and 16.64%, respectively, and enhance the PPA trade-off by 29.72%.

2.3 Critical Path Analysis

Unlike performance counters analysis [6, 18, 59] and pipeline stall analysis (interval analysis) [17, 19], the critical path analysis can answer which events we should blame for the cycle loss regarding the entire microexecution.

The critical path analysis is a methodology based on the dynamic event-dependence graph (DEG). Since its first appearance [22], it has been widely applied [24, 36, 44, 45, 56]. A DEG is a directed acyclic graph, as shown in Figure 4, with vertices denoting the pipeline stages and edges representing the dependencies. Edge weights indicate delayed cycles. The longest path from the fetch of the first instruction to the commit of the last instruction is the critical path (108 cycles, highlighted in red in Figure 4). Critical events can be obtained from the critical path, which are dependencies contributing to the critical path's length. For instance, a D-cache miss is a critical event contributing 100 cycles to the microexecution.

Although the critical path provides a global view, it is limited to clarify which resource is deficient or abundant. *Firstly, in the former DEG formulation, the dependence and weights assignment are static without adhering to actual microexecution.* This is a significant concern since microexecution contains dynamic behaviors, such as

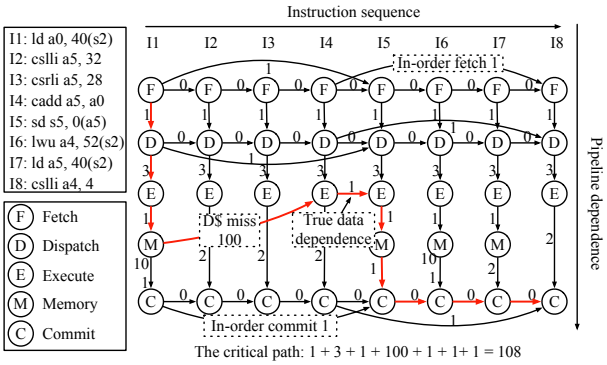


Figure 4: An overview of the dynamic event-dependence graph.

instruction scheduling and resource usage dependencies. Previous DEG formulations statically assign edges and weights following predetermined rules without considering runtime information, leading to deviations between critical path length and actual runtime. *Secondly, the critical path cannot accurately characterize the bottlenecks' contributions to the overall runtime, even if the modeled critical path length is strictly identical to the simulation runtime.* This is because the previous DEG formulation cannot distinguish overlapped events in microexecution, resulting in the double-counting of bottleneck contributions.

3 LESSONS LEARNED & DESIGN PRINCIPLES

We scrutinize the previous DEG formulation and illustrate its limitations in detail. We summarize the lessons learned and provide our design principles.

Firstly, the previous DEG formulation [22, 24, 36, 56] statically assigns weights and edges without following the actual microexecution, leading to inaccurate critical path length estimation. As shown in Figure 5(a), we demonstrate the point with 444.namd. The first error is using a static penalty to represent variant durations. For example, a branch misprediction penalty depends on the number of in-flight instructions in the wrong execution path. In Figure 5(a), the penalty of the first branch misprediction is 3 instead of 5 cycles. The error is rooted in a fundamental limitation of the former formulation, *i.e.*, the lack of events' timing in the DEG construction. The second error comes from false dependence. In Figure 5(a), the DEG formulation inserts an edge between the commit of I1 and the fetch of I9 to denote ROB dependence, contrary to the fact that no delay exists since I1 has freed the ROB entry before I9 consumes it. Although the incorrect insertion of ROB dependence is not included in the critical path (hidden by parallel events), the length of the critical path is shorter than the actual simulation time due to the lack of correct pipeline dependence delay. The ignorance of events timing information in the formulation leads to a 25.71% underestimation of the critical path.

Secondly, bottlenecks' contributions are incorrectly estimated due to the inability to distinguish concurrent events. Figure 5(b) demonstrates the third error with 456.hmmer. Consecutive execution stages are connected (E(I1) $\rightarrow$ E(I10)) to denote read/write port contention. The contention contributes nine cycles to the execution time according to the critical path. However, the contribution is four cycles in the actual microexecution. The formulation overestimates the read/write port's insufficiency by 125%. The redundant five cycles are hidden by parallel events, *e.g.*, I3 and I4 get their read/write ports simultaneously.

Embedding the events' timing information and finding a way to distinguish concurrent events is crucial for making the DEG formulation practical. This approach enables the critical path to accurately identify whether a resource is deficient or abundant. Therefore, we provide two design principles for DSE via bottleneck analysis 1) *The dependencies contributing to execution time should be captured as much as possible.* Capturing more resource usage improves the utilization approximation. 2) *Concurrent events should be distinguishable.* The distinguishability unveils whether we matter a concurrent event for bottleneck contributions to the overall execution time.

4 THE ARCHEXPLORER APPROACH

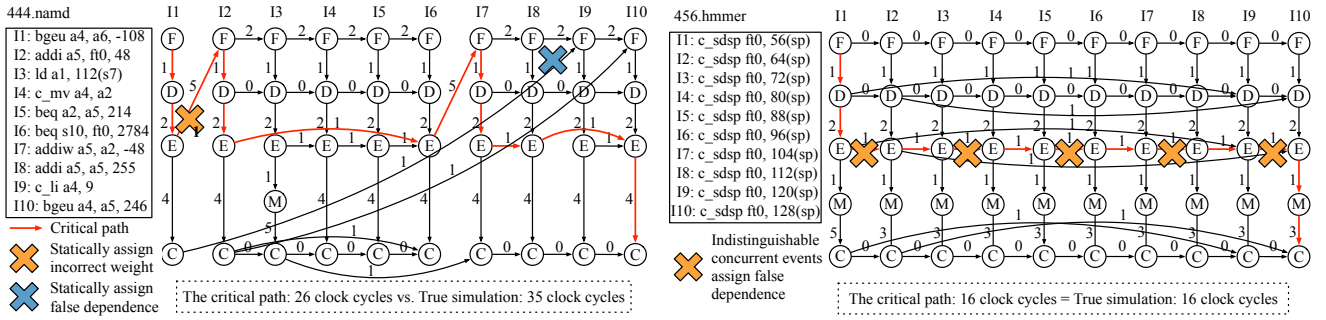
Following the design principles summarized in Section 3, we propose ArchExplorer, with an overview shown in Figure 6.

ArchExplorer involves three stages. Given the design space, the initial microarchitecture parameters are sampled or chosen with prior knowledge. In stage 1, we introduce a new DEG formulation (Section 4.1). The new formulation removes previous limitations. Based on the new DEG, we propose the induced DEG and apply critical path construction in stage 2 (Section 4.2). In stage 3 (Section 4.3), we generate a bottleneck analysis report by computing the resource contribution to the microexecution runtime. We reassign resources according to the report, producing a new design. The promising solutions are explored until the early-stopping criterion is met, and the Pareto frontier is obtained from the explored set.

4.1 New DEG Formulation of Microexecution

We propose a new DEG representation of microexecution to mitigate the limitations of the previous formulation. The backbone looks similar to the original proposal [22]. Vertices denote pipeline stages, and edges represent dependencies. However, a fundamental limitation of the former formulation is the lack of events' timing information, causing inaccurate modeling of events like speculative scheduling, resource contention, *etc.* Lacking the timing information also makes the concurrent events difficult to discriminate. We explicitly embed the events' timing information into the formulation and propose new rules to assign edges and weights dynamically. First, we give an overview of the new DEG formulation. Next, we illustrate how the new DEG is dynamically constructed. Last, we manifest how we distinguish between overlapped events.

Figure 7 is an outline of the new DEG formulation. Instead of aligning instructions with pipeline stages, as shown in Figure 4, we align instructions *w.r.t.* the time. Each vertex is accessed with two-dimensional coordinates (x, y). The X-axis denotes the timeline, and the Y-axis represents the instruction sequence. The instruction-level parallelism is shown by fixing the X coordinate. And the lifetime of an instruction is reported by fixing the Y coordinate. For edge weights, instead of using static numbers to denote, dynamic time intervals between two vertices are used in the new DEG. In



(a) Previous DEG formulation statically assigns edges and weights without following the actual microexecution.

(b) Previous DEG formulation cannot distinguish overlapped events.

Figure 5: (a) and (b) uses Calipers [24], the representative DEG formulation, to demonstrate three kinds of error sources, with critical paths highlighted in red. (a) illustrates errors including incorrect weights and false dependence due to static assignment without following actual microexecutions. (b) manifests false dependence owing to indistinguishable concurrent events.

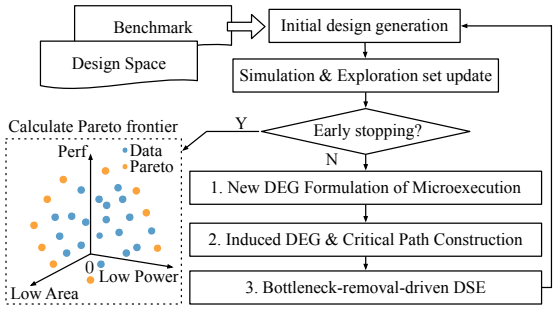


Figure 6: An overview of the ArchExplorer approach.

the following context, we use $P(I_i)$ to represent the instruction I_i's pipeline stage P [2].

We categorize four kinds of dependencies. See Table 2. The pipeline dependencies are horizontal edges. Misprediction, hardware resource, and true data dependencies are "skewed" edges, *i.e.*, they denote interactions between instructions.

- **Pipeline dependence:** It characterizes the microarchitecture pipeline implementations. More vertices are added to model architecture parameters. For example, to study the I-cache and fetch buffer size, we incorporate F1 and F2, representing when the front-end sends requests and receives replies from the I-cache. The duration of $F1(I_i) \rightarrow F2(I_i)$ equals the I-cache access latency, while F describes the moment instructions are copied to the fetch target queue. Misaligned accesses are modeled by $F2(I_i) \rightarrow F(I_i)$. To aid visualization, we merge F2 and F to formulate F2/F when the events occur simultaneously. This merging is also employed for other vertices.
- **Misprediction dependence:** It is used to model branch or memory address dependence mispredictions [13].
- **Hardware resource dependence:** The previous formulation uses the "producer-consumer" model [22] to build usage dependence on resources such as physical registers, ROB, *etc.* For example, $C(I_i) \rightarrow I(I_j)$ can represent ROB dependence, as a new ROB entry is produced at the commit stage of I_i and consumed at the issue stage of I_j. In contrast, we unify dependencies as rename to rename edges $R(I_i) \rightarrow R(I_j)$. This is because the new DEG aligns instructions strictly with time, so a "producer-consumer" edge is always assigned zero delays since newly-released resources can be used immediately. A zero delay edge prevents the critical path from capturing critical resource usage dependence. Mainstream microarchitectures often use one stage (*e.g.*, rename) to check whether required resources are available before dispatching the instruction. If a requisite resource is exhausted, a stall is incurred, otherwise, the instruction is pushed to the issue queue and waits to schedule. The rename to rename edge precisely reflects the resource usage dependence, and the time interval between these two rename stages equals the resource's duty cycles (the resource is busy during the interval).
- **True data dependence:** It characterizes the read-after-write dependence within the issue window.

Another significant difference from the previous formulation is that we construct the new DEG dynamically. That is, the new DEG is built adhering to the actual microexecution. With the time coordinate, we know whether a misprediction event occurs and how many penalties are produced. For example, if a conditional branch instruction I_i is mispredicted, microarchitecture starts squashing and refilling the pipeline with instructions in the correct execution path. Denote the first refilled instruction as I_j. We assign the edge $P(I_i) \rightarrow F1(I_j)$ once we find the time of $F1(I_j)$ is larger than $P(I_i)$ from the new DEG. The interval between $P(I_i)$ and $F1(I_j)$ is the actual squash latency due to the misprediction.

However, acquiring the pipeline stages' timing information is inadequate to identify the correct edges and weights for resource usage dependencies (such as issue queues and ROB). Although time is useful for identifying stalls, it is not sufficient to determine these edges. Which resources are preemptively occupied by which

[2] P could be F1, F2, DC, *etc.*, as shown in Figure 7. For example, $DC(I_3)$ denotes the decode stage of instruction I_3.

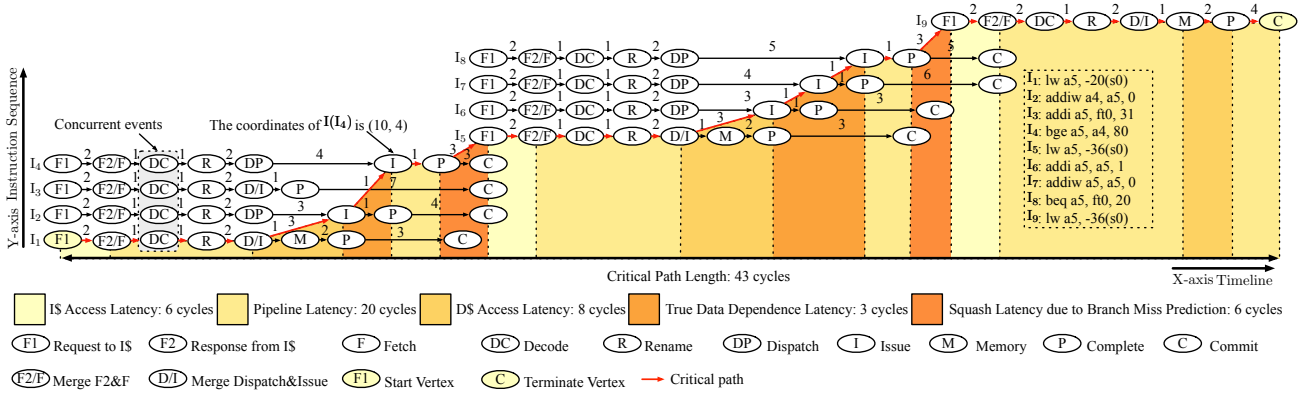


Figure 7: An overview of the new DEG formulation of microexecution. The critical path is highlighted in red. The cause of each edge in the critical path is attributed to a particular resource (shaded with colors for visualization purposes).

Table 2: The dependence specification

Type	Edge	Description
Pipeline dependence	$F1(i) \rightarrow F2(i)$	Send a request to I$, and get a response for instruction i.
	$F2(i) \rightarrow F(i)$	I$ puts the instruction i in the fetch buffer, and the fetch stage performs pre-decode or predictions.
	$F(i) \rightarrow DC(i)$	The fetch stage send instruction i to the decoder.
	$DC(i) \rightarrow R(i)$	The decode stage send μ-ops of instruction i to the rename.
	$R(i) \rightarrow DP(i)$	The rename stage send instruction i to dispatch.
	$DP(i) \rightarrow I(i)$	Schedule instruction i to issue.
	$I(i) \rightarrow P(i)$ $I(i) \rightarrow M(i) \rightarrow P(i)$	Execute instruction i with suitable functional units like ALUs or read/write ports.
	$P(i) \rightarrow C(i)$	Commit instruction i after it is finished execution.
Misprediction dependence	$P(i) \rightarrow F1(i+1)$	Instruction i encounters a branch/memory address dependence misprediction.
Hardware resource dependence	$R(i) \rightarrow R(j)$	Insufficient resources delays instruction j, and j requires those resources that instruction i releases. The edge insertion is according to the scoreboard. The resources include ROB, IQ, LQ, SQ, as well as physical integer and floating-point registers.
	$I(i) \rightarrow I(j)$	Insufficient resources delays instruction j, and j requires those resources that instruction i releases. The resources are functional units, *e.g.*, integer/floating-point ALUs, dividers, *etc.*
True data dependence	$I(i) \rightarrow I(j)$	The true data dependence. The delayed cycles are either due to D$ access or the execution of functional units.

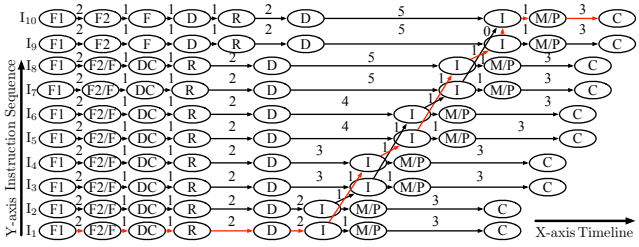


Figure 8: The new DEG formulation is applied *w.r.t.* the code snippet as shown in Figure 5(b). And it identifies the true read/write ports usage dependencies, *i.e.*, $I(I_1) \rightarrow I(I_4)$, $I(I_4) \rightarrow I(I_5)$, $I(I_5) \rightarrow I(I_8)$, and $I(I_8) \rightarrow I(I_9)$.

instructions? To address this issue, we record the correspondence between assigned resources and instructions with a scoreboard. The scoreboard uses resource entry as the granularity and indexes each entry with a unique ID number. We demonstrate how we detect and assign the correct edge with the scoreboard. Consider a microarchitecture that uses the rename stage to check whether required resources are available for each instruction, and the rename stage takes two cycles to finish. If all required resources are idle, the instruction can be dispatched immediately. We record which resource is responsible for the stall and assign a rename-to-rename edge by comparing the ID number. Specifically, the resource ID number of the dependent instruction should match that of the stalled instruction according to the scoreboard. If an instruction is stalled due to multiple insufficient resources, the rename-to-rename edges are built in turn.

With the new DEG formulation, Figure 8 elucidates how we distinguish between concurrent events *w.r.t.* the same code snippet listed in Figure 5(b). Since $I(I_1)$ and $I(I_2)$ are aligned along the X-axis, no function unit contention exists [3]. Compared to Figure 5(b), the critical path highlighted in red in Figure 8 removes duplicated read/write port dependencies.

[3] $I(I_3)$ and $I(I_4)$, $I(I_5)$ and $I(I_6)$, $I(I_7)$ and $I(I_8)$, *etc.*, are also aligned with time, meaning they are concurrent events.

4.2 Induced DEG & Critical Path Construction

The critical path is a serialization representation of parallel events. It highlights which overlapping events matter for the overall microexecution.

To facilitate the critical path construction, we introduce the *induced DEG* – a *connected* DEG (Section 4.1) consisting of horizontal, "skewed", and *virtual* edges. Horizontal edges are pipeline dependencies. "Skewed" edges (non-horizontal) signify non-pipeline dependencies, which could be resource dependencies. The virtual edges are added to make the new DEG connected. Unlike the prior DEG [22], our new DEG formulation removes consecutively connected fetch edges, commit edges, *etc.* This is because these edges are rooted in instruction execution sequences rather than resource dependencies, as they can hinder critical path construction. However, their removal may result in a disconnected graph (no path from $F1(I_1)$ to the last instruction's commit) if instructions do not have non-pipeline dependencies in highly parallel workloads. Thus, we introduce virtual edges in the induced DEG.

Formally, assume two "skewed" edges are annotated with $s_i \rightarrow e_j$ and $s_k \rightarrow e_l$ ($j < k$), where s_i, e_j, s_k, e_l are vertices in the new DEG formulation, *i.e.*, a stage of I_i, I_j, I_k, and I_l, respectively. i, j, k, and l are different instruction sequence numbers. Virtual edges could be $s_i \rightarrow s_k$ (or $e_j \rightarrow s_k$) as long as either of the following two rules is met [4].

Rule 1 **(Connect via time):** s_i is connected to s_k if the time of s_k is the closest to s_i.

Rule 2 **(Connect via instruction sequence):** s_i is connected to s_k if the instruction sequence k is the closest to i.

If multiple stages satisfy at least one of these rules, we connect them to s_k to generate more virtual edges. Virtual edges are not true dependencies but make the DEG connected. The connection of "skewed" edges allows for the exposure of consecutive resource utilization status. We connect "skewed" edges with the closest time and instruction sequence for two reasons. First, since the induced DEG is highly connected due to virtual edges, we can investigate many combinations of resource usage dependencies to formulate the critical path. Second, the closest connections allow us to directly eliminate many sub-optimal solutions, as connections that are not the closest are obviously sub-optimal according to the first design principle. The induced DEG facilitates the critical path densely composed of resource usage dependencies.

A walking example is provided in Figure 9 to demonstrate the idea more clearly. The new DEG is applied to model 11 instructions in a microexecution (Figure 9(a)), with "skewed" edges colored in orange. $R(I_1) \rightarrow R(I_{10})$ and $R(I_3) \rightarrow R(I_{11})$ denote dependencies of integer physical registers. $D/I(I_2) \rightarrow I(I_3)$, $D/I(I_5) \rightarrow I(I_6)$, and $I(I_7) \rightarrow I(I_8)$ are D-cache misses. $I(I_3) \rightarrow I(I_4)$ and $I(I_7) \rightarrow I(I_8)$ are true data dependencies due to register a_5. $P(I_1) \rightarrow F1(I_2)$ represents a branch misprediction. Although I_9 and I_{10} have a true data dependence due to register a_5, there is no stall in actual microexecution because I_9 and I_{10} are not in the same issue window (I_{10} is not dispatched at the time of $D/I(I_9)$). $F1(I_1)$ is not reachable to $C(I_{11})$. Figure 9(b) illustrates the corresponding induced DEG with virtual edges colored in blue. Specifically, we use $R(I_1) \rightarrow R(I_3)$ and $R(I_1) \rightarrow D/I(I_2)$ as two examples to elucidate rules 1 and 2. In Figure 9(a), a virtual edge $R(I_1) \rightarrow R(I_3)$ is created as $R(I_3)$'s time is the closest to $R(I_1)$ among all "skewed" edges, and I_3 and I_1 are different instructions (rule 1). Similarly, a virtual edge $R(I_1) \rightarrow D/I(I_2)$ is created since the instruction sequence of $D/I(I_2)$ is the closest to that of $R(I_1)$ among all "skewed" edges (rule 2).

[4]We use $s_i \rightarrow s_k$ as an example. The rules are the same for $e_j \rightarrow s_k$.

Algorithm 1 Critical Path Construction

Require: G: The induced DEG with the edge cost;
1: node = `topological_sort`(G);
2: Initialize edge cost vector $\boldsymbol{d}$ with all zero;
3: Initialize the path vector $\boldsymbol{p}$ with all zero;
4: **for** $n \leftarrow$ node **do**
5: **if** $\mathcal{N}_G(n) \neq \emptyset$ **then** ▷ $\mathcal{N}_G(n)$ are predecessors of n.
6: $\boldsymbol{d}[n] = \arg \max_{v \in \mathcal{N}_G(n)} d[v] + \text{cost}$; assign $\boldsymbol{p}[n]$ with v;
7: **else**
8: $\boldsymbol{d}[n] = 0$; $\boldsymbol{p}[n] = n$;
9: **end if**
10: **end for**
11: **return** `reverse`($\boldsymbol{p}$);

Although the induced DEG connects the graph, it cannot answer which concurrent event should be blamed for the microexecution. We propose a dynamic programming-based critical path construction algorithm to decide accordingly. To capture more resource usage dependencies and not miss "important" edges, we define *costs* to edges as follows: (1) All horizontal edges have zero cost. (2) All virtual edges have zero cost. (3) All true data dependencies have zero cost. (4) All "skewed" edges, except true data dependencies, have costs equal to the interval between two vertices. We set horizontal edges with zero cost to capture the true resource usage dependence rather than pipeline stalls. Using the induced DEG with edge costs, we apply the longest path search with linear time complexity in Algorithm 1.

In Algorithm 1, line 6 uses dynamic programming to record the temporary longest path to n. The critical path can be obtained by reversing $\boldsymbol{p}$ at line 11. For the example in Figure 9(b), the critical path is $F1(I_1) \rightarrow F2/F(I_1) \rightarrow DC(I_1) \rightarrow R(I_1) \rightarrow R(I_{10}) \rightarrow C(I_{11})$. From the critical path, we discover that the top bottleneck is the insufficiency of physical integer registers, and the branch misprediction cannot hide it. The length of the critical path is precisely the same as the simulation time.

4.3 Bottleneck-removal-driven DSE

We conduct the bottleneck-removal-driven DSE by reassigning resources via eliminating bottlenecks gradually. Scarce resources are increased to mitigate performance bottlenecks, while over-provisioned resources are reduced to balance power and area. Identifying deficient or abundant resources is achieved by computing their contribution to the overall runtime through the constructed critical path. The higher the contribution, the scarcer the resources and the greater the necessity to assign more of them. Conversely, redundant resource leads to zero contribution and can be appropriately reduced.

The main idea of computing the resource contribution is to attribute the cause of each edge in the critical path to a particular

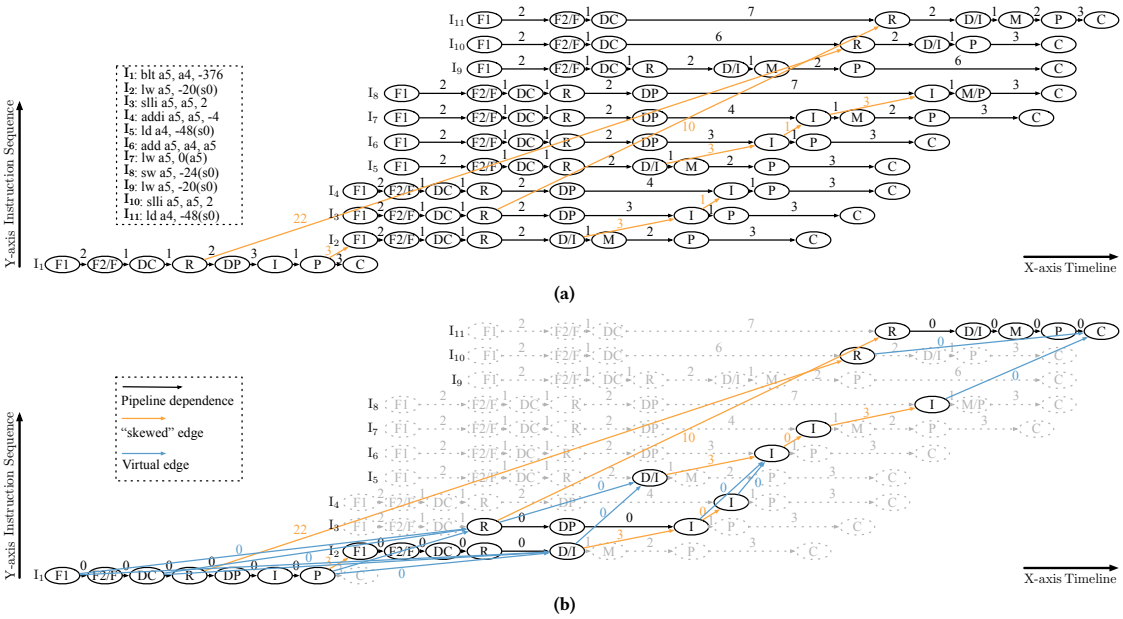


Figure 9: (a) An example code snippet and its corresponding new DEG formulation. (b) The overview of induced DEG with edge cost extracted from DEG.

resource. This attribution is straightforward, as we define the meaning of "skewed" edges in Table 2. We do not attribute virtual edges to resources. Figure 7 illustrates some attribution highlighted with diverse colors, where each edge is a non-overlapping segment in the microexecution. We introduce the resource contribution in two cases and describe our resource assignment strategies.

Formally, for a critical path p with length L and containing N (non-overlapping) edges, a resource b's contribution $c(b)$ is computed according to Equation 1,

$$c(b) = \sum_{i=1}^{N} l_i \mathbb{1}[p(i) = b]/L, \tag{1}$$

where $\mathbb{1}(\cdot)$ is the indicator function, which outputs 1 if its argument is true and 0 otherwise. l_i is the delay (not edge cost as referred to in Section 4.2) of the i-th edge $p(i)$, attributed to the resource b. We determine the type of resource according to $c(b)$, *i.e.*, whether the resource is top-ranked deficient or abundant. For multiple workload evaluations, as shown in Equation 2, we do a weighted average for the contributions of each resource.

$$\bar{c}(b) = \sum_{i=1}^{|B|} w_i \cdot c_i(b), \quad \sum_{i=1}^{|B|} w_i = 1, \tag{2}$$

where $|B|$ is the number of workloads, and $c_i(b)$ is computed from Equation 1. The symbol $\bar{c}(b)$ is the average resource contribution, where w_i is the i-th weighted coefficient for a specific workload that encodes the designer's preference. Each resource's contribution is summarized in an output report. We use $\bar{c}(b)$ to guide the DSE procedure. The reassigned parameter values are decided based on the design space specification. Specifically, we select the next larger candidate value from the specification if we need to increase the value, and we decrease it to the next smaller candidate value if such resources do not have a contribution.

Unlike resources such as ROB entries, branch predictors and caches are special. Assigning more resources may not decrease their contributions, as benchmarks contain hard-to-predict branches or specific data access patterns that are fundamental to the prediction algorithm. Increasing the branch target buffer (BTB), return address stack (RAS), *etc.*, may not improve the branch predictor. Similarly, larger capacities and associativities may not remove caches' contributions on complex access patterns. We argue that performance cannot be effectively enhanced by only reassigning more resources to them. The solution to the problem requires a suitable branch prediction algorithm or a better cache replacement policy. We stop reassigning more resources to branch predictors and caches in the DSE if the PPA improvement is limited.

Figure 10 provides an overview of a search path conducted by ArchExplorer. In the second step, the report indicates that the microarchitecture lacks sufficient store queues (SQs), which contribute 38% to the execution time. Thus, in the third step, more SQs are assigned, resulting in an 8% alleviation of the bottleneck. In the fourth step, increasing SQs further mitigates the bottleneck, while decreasing redundant resources saves the area overhead. The DSE uncovers the reasons for PPA improvements stepwise, highlighting potential optimization opportunities for the load-store unit, with assigning more SQs being one optimization. Architects can gain valuable insights into acquiring good solutions gradually. Black-box methods cannot offer the same merit.

ArchExplorer reassigns resources until the maximum search budget or no further PPA trade-off improvement. It restarts to explore with a new microarchitecture. Explored designs are recorded in an exploration set for Pareto frontier computation (Figure 6).

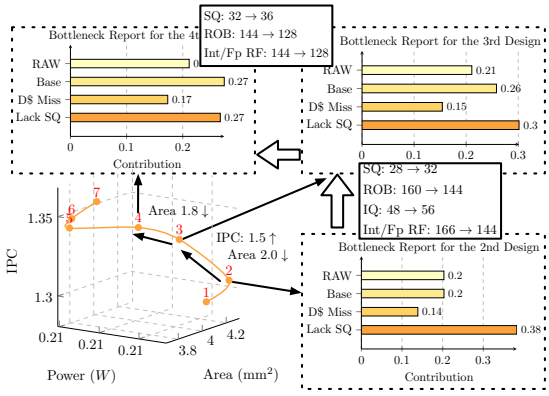


Figure 10: An overview of a search path.

5 EXPERIMENTAL SETUP & EVALUATION METRICS

5.1 Simulation Environment

We use GEM5 [9, 39], as the timing-accurate simulator and McPAT [38] as the power and area modeling tool. We modify GEM5 to generate dynamic timing information and implement ArchExplorer with C++ and Python. SPEC CPU2006 [2, 25] (SPEC06) and SPEC CPU2017 [3] (SPEC17) are utilized in benchmark evaluations, as listed in Table 3. We use Simpoints [47] of each workload to evaluate. Each Simpoint includes a hundred million instructions, warming up using ten million instructions. Since identifying the resource utilization status does not require the entire workload, we use the first hundred thousand instructions of each Simpoint to calculate the critical path with ArchExploer. After the DSE, the explored Pareto solutions are re-evaluated with the full Simpoints.

We implement ArchRanker [12], AdaBoost [37], and BOOM-Explorer [8] as baselines. ArchRanker [12] leverages a black-box model for ranking. AdaBoost [37] also adopts machine-learning techniques [20, 52, 54]. BOOM-Explorer [8] uses Bayesian optimization [14] with active-learning [61]. The baselines represent recent solutions using advanced machine-learning techniques. We also compare with Calipers [24] since it uses the previous DEG formulation to enhance fast DSE. Although some baselines, like ArchRanker [12] and BOOM-Explorer [8], target different out-of-order (OoO) processors, their methods are transferable.

The design space of an OoO RISC-V processor similar to Alpha 21264 [33] is listed in Table 4. The values of the third column are "start number:end number:stride". A two-level cache hierarchy with a 16GB DRAM main memory is applied. The first level cache is for optimization, while the L2 cache is 8-way associative with 2MB. Diverse components like branch prediction units, functional units, and L1 cache structures, *etc.*, are included. The design space size is more than 8.9×10^{14}. For ArchExplorer, we stop the optimization until we find that the PPA trade-off curve starts to plateau. We generate the initial design randomly. Baseline microarchitectures suggested by architects can also be leveraged in the initialization stage to achieve better results. All weighted coefficients w_i in Equation 2 are $1/|B|$ since we target average cases and assume no preference for workloads used in experiments.

Table 3: Workloads used for evaluation

Benchmark suite	# of Workloads	Example Workloads
SPEC06	12	bzip2, namd, dealII, h264ref, soplex, povray, ...
SPEC17	14	perlbench_s, gcc_s, cactuBSSN_s, nab_s, ...

Table 4: Microarchitecture design space specification

Components	Description	Hardware Resource	#
Pipeline width	fetch/decode/rename/dispatch/ issue/writeback commit width	1:8:1	8
Fetch buffer	fetch buffer size in bytes	16, 32, 64	3
Fetch queue	fetch queue size in μ-ops	8:48:4	11
Local predictor	local predictor size of the Tournament BP	512, 1024, 2048	3
Global/Choice predictor	global predictor size of the Tournament BP	2048, 4096, 8192	3
RAS	return address stack size	16:40:2	13
BTB	branch target buffer size	1024, 2048, 4096	3
ROB	reorder buffer entries	32:256:16	15
Int RF	number of physical integer registers	40:304:8	18
Fp RF	number of physical floating-point registers	40:304:8	18
IQ	number of instruction queue entries	16:80:8	9
LQ	number of load queue entries	20:48:4	8
SQ	number of store queue entries	20:48:4	8
IntALU	number of integer ALUs	3:6:1	4
IntMultDiv	number of integer multipliers and dividers	1, 2	2
FpALU	number of floating-point ALUs	1, 2	2
FpMultDiv	number of floating-point multipliers and dividers	1, 2	2
I$ size	the size of I$ in KB	16, 32, 64	3
I$ assoc.	associative sets of I$	2, 4	2
D$ size	the size of D$ in KB	16, 32, 64	3
D$ assoc.	associative sets of D$	2, 4	2
Total size	8.9649×10^{14}		

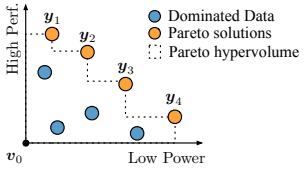


Figure 11: The visualization of Pareto hypervolume in Perf-Power space. Pareto hypervolume is the area bounded by $\mathcal{P}(\mathcal{Y}) = \{\boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{y}_3, \boldsymbol{y}_4\}$ and the reference point $\boldsymbol{v}_0$.

5.2 Evaluation Metrics

Since we target multiple objectives involving higher performance, power efficiency, and area efficiency, we use *Pareto hypervolume* as the evaluation metric to compare different DSE algorithms fairly. Pareto hypervolume is a widely applied metric to measure how good the explored Pareto frontier is [53]. Denote a Pareto frontier as $\mathcal{P}(\mathcal{Y}) = \{\boldsymbol{y}_1, \boldsymbol{y}_2, ..., \boldsymbol{y}_n\} \subseteq \mathrm{Y}$, with $\boldsymbol{y}_i$ representing PPA values of i-th design, and Y is the objective space. A reference point $\boldsymbol{v}_0$ is set and dominated by $\mathcal{P}(\mathcal{Y})$, *i.e.*, $\forall \boldsymbol{y}_i \in \mathcal{P}(\mathcal{Y})$, the PPA values are all better than $\boldsymbol{v}_0$, and we denote $\boldsymbol{y}_i \succeq \boldsymbol{v}_0$. Pareto hypervolume is

defined as

$$\mathrm{PV}_{\boldsymbol{v}_0}(\mathcal{P}(\mathcal{Y})) = \int_{\mathrm{Y}} \mathbb{1}[\boldsymbol{y} \succeq \boldsymbol{v}_0][1 - \prod_{\boldsymbol{y}_* \in \mathcal{P}(\mathcal{Y})} \mathbb{1}[\boldsymbol{y}_* \not\succeq \boldsymbol{y}]]\mathrm{d}\boldsymbol{y}, \quad (3)$$

where the integral sums the space bounded from $\boldsymbol{v}_0$ to $\mathcal{P}(\mathcal{Y})$ [53], as shown in Figure 11. We also include the number of simulations spent as another evaluation metric. The higher the Pareto hypervolume and the fewer the simulations, the better the DSE algorithm. We measure the PPA trade-off as $\mathrm{Perf}^2/(\mathrm{Power} \times \mathrm{Area})$.

6 RESULTS

6.1 Comparison w. DSE Methodologies

Figure 12 visualizes the Pareto hypervolume curves in terms of simulations for each algorithm. The average PPA values among workloads are used in Pareto hypervolume computing. The Pareto hypervolume is non-decreasing since more Pareto designs are explored as simulations continue. It is worth noting that ArchExplorer achieves higher Pareto hypervolume very early, and dominates other methods at different simulation budgets. Two cases are selected for comparison to study the improved benefits. First, how many simulations [5] are needed when achieving a target Pareto hypervolume? Second, how much Pareto hypervolume can be attained when all methods use the same simulation budgets? We choose $y = 15.80$ and $x = 3000$ for SPEC06 and $y = 15.60$ and $x = 2400$ for SPEC17 as two cases, as shown in Figure 12. We chose these two cases for SPEC06 and SPEC17 because, at this moment, the performance curves tend to converge. The corresponding results are shown in Table 5. In SPEC06, compared to ArchRanker [12], AdaBoost [37], and BOOM-Explorer [8], ArchExplorer users 14.47% more, 24.56% fewer, and 74.12% fewer simulations when $y = 15.80$, respectively. For $x = 3000$, the gained Pareto hypervolume of ArchExplorer surpasses BOOM-Explorer [8], AdaBoost [37], and ArchRanker [12] by 1.58%, 4.20%, and 3.32%, respectively. In SPEC17, ArchExplorer can save 74.63% of simulation budgets at most and achieve 6.80% higher Pareto hypervolume. Particularly, the relatively modest increases in Pareto hypervolume translate into significant performance improvements. For example, when the simulation budget equals 480 in SPEC17, the Pareto hypervolume of ArchExplorer is improved with an average of 5.40% than all AdaBoost [37]. However, the performance of Pareto designs is higher up to 16.20%. The results demonstrate that ArchExplorer can achieve comparable solution qualities by removing large simulation budgets. ArchRanker [12] compares pairs of designs to determine which is better and conducts the constrained DSE with a binary search. Due to the complicated design space introduced in Section 2.1, the trained ranking model is hard to give accurate rankings when confronting a large design space and complex workloads.

To figure out how ArchExplorer performs better than black-box methods, we plot the Pareto frontiers of all methods with 3600 simulations in SPEC06, as shown in Figure 13 [6]. In IPC-1/Power space, the Pareto frontiers of each method are relatively close. However, in IPC-1/Area and Area/Power space, ArchExplorer's Pareto

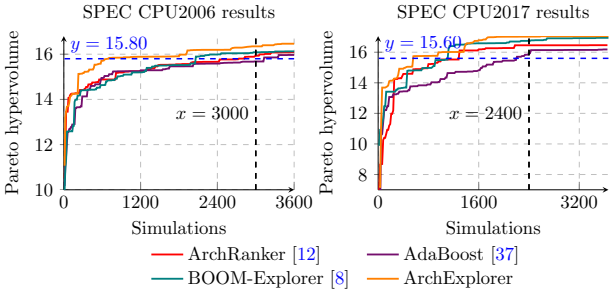


Figure 12: The visualization of Pareto hypervolume curves in terms of the number of simulations.

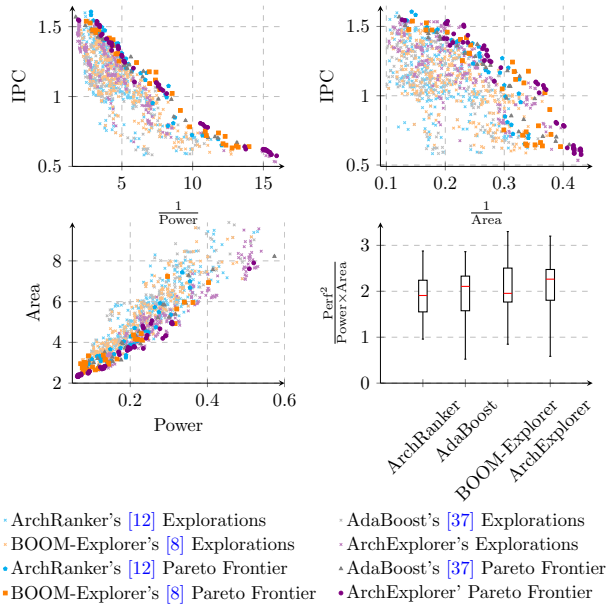


Figure 13: The visualization of Pareto frontiers and the distributions of PPA trade-offs for all methods.

frontier dominates other methods in several regions, particularly close to the origin of Area/Power space. The visualization suggests that ArchExplorer outperforms other methods not by exploring a greater number of higher-performance microarchitectures but by higher power and area efficiency designs. Figure 13 also shows the PPA trade-off distributions of Pareto designs for each method. ArchExplorer's Pareto designs achieve an average of 2.26 in the trade-off, surpassing BOOM-Explorer [8], AdaBoost [37], and ArchRanker [12] by 15.81%, 7.47%, and 18.63%, respectively. These results demonstrate that we can achieve a better PPA trade-off by assigning and removing indispensable hardware resources based on their performance contribution.

Why can ArchExplorer surpass baselines? ArchExplorer can surpass DSE methods with black-box models because previous methods [8, 12, 37] rely on AdaBoost.RT [37], Gaussian process [8], *etc.*, to learn relationships between microarchitecture features and PPA values. These models can be trained before or during DSE, but they are purely algorithm-driven and do not consider

[5] Compared to Calipers, the induced DEG has an average of 39.59% more vertices and 51.72% fewer edges with SPEC17 Simpoints. In our experiments, the longest path evaluation in ArchExplorer incurs 2.24% of the simulation runtime, which can be negligible.

[6] Due to the page limit, the Pareto frontiers in SPEC17 are not shown.

Table 5: Comparison under two cases.

Methods	SPEC CPU2006				SPEC CPU2017			
	Pareto hypervolume at $y = 15.80$		# of Simulations at $x = 3000$		Pareto hypervolume at $y = 15.60$		# of Simulations at $x = 2400$	
	# of Simulations	Ratio	Pareto hypervolume	Ratio	# of Simulations	Ratio	Pareto hypervolume	Ratio
ArchRanker [12]	2736	1	15.9185	1	1296	1	16.4542	1
AdaBoost [37]	3132	1.1447	15.6785	0.9849	2208	0.7037	15.9359	0.9685
BOOM-Explorer [8]	2064	0.7544	16.0854	0.0105	1120	0.8642	16.7416	1.0175
ArchExplorer	708	0.2588	16.3473	1.0269	560	0.4321	17.0198	1.0344

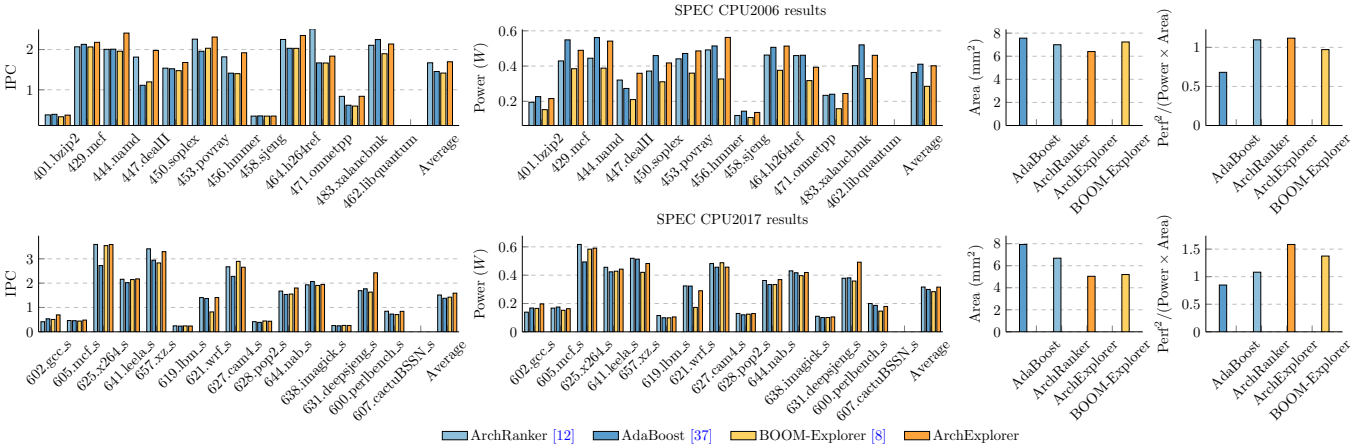


Figure 14: Comparisons between the Pareto designs in performance and power.

microarchitecture analysis in depth. As a result, more simulations are required in the DSE. In contrast, ArchExplorer applies domain knowledge in DSE directly by identifying bottlenecks and adjusting hardware resources to eliminate them, resulting in a better PPA balance immediately. In summary, ArchExplorer's explainable DSE process can provide insights for architects that may not be available through an AI model.

6.2 Comparison w. Best Balanced Designs

To study how high-performance design balances PPA well, we rank the designs explored by each method with $\text{Perf}^2/(\text{Power} \times \text{Area})$ and select the ones with the highest performance for comparison. Figure 14 lists the results for SPEC06 and SPEC17. Table 6 lists key parameters of each Pareto design [7]. For SPEC06, on average, ArchExplorer's best balanced design achieves 1.53%, 16.65%, and 19.81% higher performance than ArchRanker [12], AdaBoost [37], and BOOM-Explorer [8]. It also saves power by 2.15% compared to AdaBoost's [37]. ArchExplorer improves the area by an average of 11.79%. Namely, ArchExplorer's Pareto design is better than other methods by an average of 56.05% and, at most, 64.29% in the PPA trade-off. In SPEC17, ArchExplorer's solution achieves an average of 9.46% higher performance. While the solution sacrifices 5.54% more power compared to baselines, it attains a 20.07% smaller area and 49.53% higher PPA trade-off. Although the designs in Table 6 look similar, big differences in the PPA trade-off are observed. The results illustrate that better solutions are achieved by balancing resources. For example, the ROB should not be allocated many entries compared to ArchRanker [12]. Otherwise, it can degrade the overall performance by long squashing due to misprediction events. Redundant resources like floating-point physical registers can worsen power dissipation compared to AdaBoost [37].

[7] Due to the page limit, we do not list Pareto designs for SPEC17.

6.3 Comparison w. Calipers

Calipers [24] represents the latest critical path analysis method to enable fast DSE. By scanning the design space, Calipers [24] assists in exploring high-performance microarchitectures. We use a different experimental setup to demonstrate the superiority in identifying the type of resources with the new DEG formulation over Calipers [24]. Calipers [24] only targets performance and neglects the consideration for power and area optimization. And it does not provide how to search except for applying the previous DEG formulation in performance modeling with microexecution. On the other hand, Calipers [24] can model more accurately with information like branch prediction results and cache access penalties from the simulation. Hence, we extract very similar 1296 designs from Table 4 and use Calipers [24] to sweep the design space and retrieve the highest-performance solution. The reasons are twofold. First, it helps Calipers to obtain what it considers the global optimal solution for a fair comparison. Second, very similar designs can expose modeling differences between two DEG formulations in deep. We apply ArchExplorer to the same sub-design space and compare the solutions in performance and power. Different from Calipers [24], ArchExplorer does not scan the entire design space.

Results are shown in Figure 15. The solution found by ArchExplorer outperforms Caliper's [24] by 2.11% in performance on

Table 6: Key parameters of Pareto designs

Components	Method			
	AdaBoost [37]	ArchRanker [12]	BOOM-Explorer [8]	ArchExplorer
Pipeline width	8	8	8	8
ROB/IQ	176/56	192/72	112/44	176/72
Int/Fp RF	192/208	256/96	240/192	240/80
LQ/SQ	44/48	48/44	24/40	48/48
IntALU IntMultDiv	5/1	4/2	4/2	4/2
FpALU FpMultDiv	2/1	2/1	2/2	1/1
I$	4-way 64K	2-way 64K	2-way 64K	4-way 64K
D$	4-way 64K	2-way 64K	2-way 32K	4-way 64K

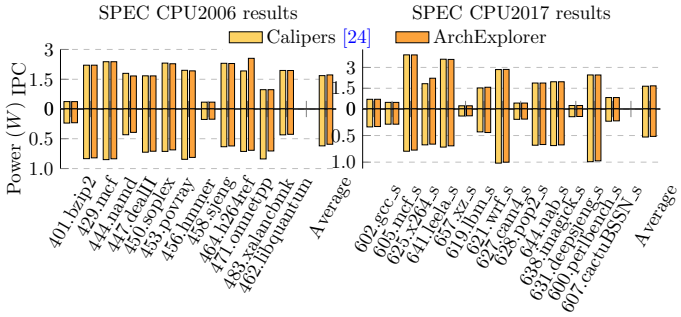


Figure 15: Comparisons w. Calipers [24].

average in SPEC06. And ArchExplorer's solution achieves 4.36% lower power and 2.38% lower area. In SPEC17, we receive a 1.88% higher performance compared to Calipers [24]. The previous DEG formulation incorrectly modeled the microexecution, leading to sub-optimal solutions. Calipers cannot identify which microarchitecture achieves higher performance with a very small ROB resource difference. Conversely, ArchExplorer adopts the new DEG formulation and attains better results by identifying the deficient resources more accurately. Notably, compared to Calipers [24] which traverses the entire design space using thousands of simulations, the improved performance benefits are gained by only using 48 simulations in ArchExplorer for SPEC06 and SPEC17, respectively. In summary, due to the more accurate and practical new DEG formulation, ArchExplorer performs better than Calipers [24].

7 DISCUSSIONS

The new DEG formulation can set up many research opportunities for microarchitecture research.

Combine with machine learning (ML): The new DEG formulation provides richer features for recent powerful deep learning models such as graph neural networks [51, 58]. Performance modeling is possible by combining features extracted from vertices and edges. Combining ML techniques and the new DEG formulation can also improve the DSE procedure.

Instruction scheduling: The instruction-level parallelism can be modeled by projecting "skewed" edges to the Y-axis. The projection length equals the degree of parallelism. The information helps design new criticality-driven instruction scheduling algorithms for new emerging workloads.

Multi-core formulation: Commodity microprocessors continue to scale in a number of cores (96 cores in AMD EPYC Genoa [41], 64 Intel Raptor Cove in Emerald Rapids [4]), given the technology scaling. Hence, the DEG formulation for multi-core deserves exploitation. It is beneficial to find bottlenecks for a multi-core system.

8 ADDITIONAL RELATED WORK

Critical Path Analysis. Fields *et al.* are the first to propose the critical path analysis with DEG to unveil bottlenecks in microexecution [22]. This approach has been broadly used [21, 23, 24, 36, 42, 44, 45, 50, 56]. Behnam *et al.* adopt it to remove bottlenecks in uniprocessors [50]. Nowatzki *et al.* extend Fields' model and propose the transformable dependence graph (TDG) [44]. The TDG is further leveraged in modeling heterogeneous accelerators [45]. However, assigning dependencies and weights without adhering to actual microexecution in TDG leads to inaccurate resource contention modeling, as also mentioned in their work [45]. Lee *et al.* [36] and Calipers [24] enable fast DSE for microarchitecture parameters also based on Fields' model. Compared to the prior works [24, 36], ArchExplorer points out the source of error for previous DEG formulations and proposes the design principles and implementations accordingly. The new DEG formulation eliminates the limitations of Fields' model.

Microarchitecture Design Space Exploration. Many works [8, 10, 12, 15, 24, 28, 31, 32, 35–37, 40, 46] proposed solutions to DSE for microarchitecture parameters. They either leverage massive access to expert knowledge [19, 30, 31] or require high computing resources to train a black-box model [8, 12, 37] to conduct the DSE. Compared to Karkhanis and Smith [31], ArchExplorer circumvents the experts' efforts by adopting an automated bottleneck analysis. Compared to prior black-box methodologies [8, 12, 28, 35, 37], ArchExplorer conducts DSE by uncovering the bottlenecks rather than relying on black-box models.

9 CONCLUSION

In this paper, we discuss the problem of DSE for microarchitecture parameters. To alleviate massive domain knowledge requirements for mechanistic models and to reduce high computing demands for black-box methods, we propose ArchExplorer, an automated bottleneck analysis-driven DSE approach implementing two design principles. We propose a new DEG formulation. An optimal critical path construction algorithm is also proposed to capture hardware resource utilization status. Several resource reassignment strategies are leveraged to remove bottlenecks and trade-off PPA values. ArchExplorer achieves an average of 6.80% Pareto hypervolume improvement and reduces more than 74.63% simulation overheads compared to previous state-of-the-art solutions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of MICRO 2023 for their encouraging feedback. This research was partially supported by ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by *InnoHK* funding, Hong Kong SAR.

REFERENCES

[1] 2013. Official RISC-V Benchmark Suites. https://github.com/riscv-software-src/riscv-tests.

[2] 2018. SPEC CPU 2006. https://www.spec.org/cpu2006/.

[3] 2022. SPEC CPU 2017. https://www.spec.org/cpu2017/.

[4] 2023. Intel Emerald Rapids. https://en.wikipedia.org/wiki/Emerald_Rapids.

[5] 2023. Tenstorrent RISC-V OoO Superscalar Processor Family. https://tenstorrent.com/risc-v/.

[6] Jennifer M Anderson, Lance M Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R Henzinger, Shun-Tak A Leung, Richard L Sites, Mark T Vandevoorde, Carl A Waldspurger, and William E Weihl. 1997. Continuous Profiling: Where Have All The Cycles Gone? *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 357–390.

[7] Omid Azizi, Aqeel Mahesri, Benjamin C Lee, Sanjay Jeram Patel, and Mark Horowitz. 2010. Energy-performance Tradeoffs in Processor Architecture and Circuit Design: a Marginal Cost Analysis. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. 26–36.

[8] Chen Bai, Qi Sun, Jianwang Zhai, Yuzhe Ma, Bei Yu, and Martin DF Wong. 2021. BOOM-Explorer: RISC-V BOOM Microarchitecture Design Space Exploration Framework. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–9.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The GEM5 Simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[10] David Brooks, Pradip Bose, Viji Srinivasan, Michael K Gschwind, Philip G Emma, and Michael G Rosenfield. 2003. New Methodology for Early-stage, Microarchitecture-level Power-performance Analysis of Microprocessors. *IBM Journal of Research and Development* 47, 5.6 (2003), 653–670.

[11] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *IEEE International Conference on High Performance Computing (HiPC)*. 1–12.

[12] Tianshi Chen, Qi Guo, Ke Tang, Olivier Temam, Zhiwei Xu, Zhi-Hua Zhou, and Yunji Chen. 2014. ArchRanker: A Ranking Approach to Design Space Exploration. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. IEEE.

[13] George Z Chrysos and Joel S Emer. 1998. Memory Dependence Prediction Using Store Sets. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. IEEE, 142–153.

[14] Samuel Daulton, Maximilian Balandat, and Eytan Bakshy. 2020. Differentiable Expected Hypervolume Improvement for Parallel Multi-objective Bayesian Optimization. *Annual Conference on Neural Information Processing Systems (NIPS)* 33 (2020), 9851–9864.

[15] Christophe Dubach, Timothy Jones, and Michael O'Boyle. 2007. Microarchitectural Design Space Exploration Using an Architecture-centric Approach. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 262–271.

[16] Christophe Dubach, Timothy M Jones, and Michael FP O'Boyle. 2008. Exploring and predicting the architecture/optimising compiler co-design space. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 31–40.

[17] Philip G Emma. 1997. Understanding Some Simple Processor-performance Limits. *IBM Journal of Research and Development* 41, 3 (1997), 215–232.

[18] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. 2006. A Performance Counter Architecture for Computing Accurate CPI Components. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* 41, 11 (2006), 175–184.

[19] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. 2009. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)* 27, 2 (2009), 1–37.

[20] Kai-Tai Fang and Yuan Wang. 1993. *Number-theoretic Methods In Statistics*. Vol. 51. CRC Press.

[21] Brian Fields, Rastislav Bodik, and Mark D Hill. 2002. Slack: Maximizing Performance Under Technological Constraints. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. IEEE, 47–58.

[22] Brian Fields, Shai Rubin, and Rastislav Bodik. 2001. Focusing Processor Policies via Critical-path Prediction. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. IEEE, 74–85.

[23] Brian A Fields, Rastislav Bodik, Mark D Hill, and Chris J Newburn. 2003. Using Interaction Costs for Microarchitectural Bottleneck Analysis. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 228–239.

[24] Hossein Golestani, Rathijit Sen, Vinson Young, and Gagan Gupta. 2022. Calipers: A Criticality-aware Framework for Modeling Processor Performance. *ACM International Conference on Supercomputing (ICS)* (2022).

[25] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[26] Mark D Hill and Alan Jay Smith. 1989. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.* 38, 12 (1989), 1612–1630.

[27] MS Hrishikesh, Norman P Jouppi, Keith I Farkas, Doug Burger, Stephen W Keckler, and Premkishore Shivakumar. 2002. The Optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. IEEE, 14–24.

[28] Engin İpek, Sally A McKee, Rich Caruana, Bronis R de Supinski, and Martin Schulz. 2006. Efficiently Exploring Architectural Design Spaces Via Predictive Modeling. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* 40, 5 (2006), 195–206.

[29] Norman P. Jouppi. 1989. The Nonuniform Distribution of Instruction-level and Machine Parallelism and Its Effect on Performance. *IEEE Trans. Comput.* 38, 12 (1989), 1645–1658.

[30] Tejas S Karkhanis and James E Smith. 2004. A First-order Superscalar Processor Model. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. IEEE, 338–349.

[31] Tejas S Karkhanis and James E Smith. 2007. Automated Design of Application Specific Superscalar Processors: An Analytical Approach. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. 402–411.

[32] Vinod Kathail, Shail Aditya, Robert Schreiber, B Ramakrishna Rau, Darren C Cronquist, and Mukund Sivaraman. 2002. PICO: Automatically Designing Custom Computers. *IEEE Trans. Comput.* 35, 9 (2002), 39–47.

[33] Richard E Kessler. 1999. The Alpha 21264 Microprocessor. *IEEE Micro* 19, 2 (1999), 24–36.

[34] Balasubramanian Kumar and Edward S. Davidson. 1980. Computer System Design Using a Hierarchical Approach to Performance Evaluation. *Commun. ACM* 23, 9 (1980), 511–521.

[35] Benjamin C Lee and David M Brooks. 2007. Illustrative Design Space Studies with Microarchitectural Regression Models. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 340–351.

[36] Jaewon Lee, Hanhwi Jang, and Jangwoo Kim. 2014. RpStacks: Fast and Accurate Processor Design Space Exploration Using Representative Stall-event Stacks. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 255–267.

[37] Dandan Li, Shuzhen Yao, Yu-Hang Liu, Senzhang Wang, and Xian-He Sun. 2016. Efficient Design Space Exploration Via Statistical Sampling and AdaBoost Learning. In *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[38] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 469–480.

[39] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The GEM5 Simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[40] Mayan Moudgill, J-D Wellman, and Jaime H Moreno. 1999. Environment for PowerPC Microarchitecture Exploration. *IEEE/ACM International Symposium on Microarchitecture (MICRO)* 19, 3 (1999), 15–25.

[41] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H Loh, Mahesh Subramony, and Sean White. 2021. Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families: Industrial Product. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. IEEE, 57–70.

[42] Ramadass Nagarajan, Xia Chen, Robert G McDonald, Doug Burger, and Stephen W Keckler. 2006. Critical Path Analysis of the TRIPS Architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 37–47.

[43] Derek B Noonburg and John P Shen. 1994. Theoretical Modeling of Superscalar Processor Performance. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE.

[44] Tony Nowatzki, Venkatraman Govindaraju, and Karthikeyan Sankaralingam. 2015. A Graph-based Program Representation for Analyzing Hardware Specialization Approaches. *IEEE Computer Architecture Letters (CAL)* 14, 2 (2015), 94–98.

[45] Tony Nowatzki and Karthikeyan Sankaralingam. 2016. Analyzing Behavior Specialized Acceleration. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* 51, 4 (2016), 697–711.

[46] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. 2009. ReSPIR: A Response Surface-based Pareto Iterative Refinement For Application-specific Design Space Exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 28, 12 (2009), 1816–1829.

[47] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. *ACM SIGMETRICS Performance Evaluation Review* 31, 1 (2003), 318–319.

[48] Chris H Perleberg and Alan Jay Smith. 1993. Branch Target Buffer Design and Optimization. *IEEE Trans. Comput.* 42, 4 (1993), 396–412.

[49] Laurence J Peter, Raymond Hull, et al. 1969. *The Peter Principle*. Vol. 4. Souvenir Press London.

[50] Behnam Robatmili, Sibi Govindan, Doug Burger, and Stephen W Keckler. 2011. Exploiting Criticality to Reduce Bottlenecks in Distributed Uniprocessors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 431–442.

[51] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The Graph Neural Network Model. *IEEE Transactions on Neural Networks (TNN)* 20, 1 (2008), 61–80.
[52] Robert E Schapire. 2003. The Boosting Approach to Machine Learning: An Overview. *Nonlinear estimation and classification* (2003), 149–171.
[53] Amar Shah and Zoubin Ghahramani. 2016. Pareto Frontier Learning with Expensive Correlated Objectives. In *International Conference on Machine Learning (ICML)*. PMLR, 1919–1927.
[54] Durga L Shrestha and Dimitri P Solomatine. 2006. Experiments with AdaBoost. RT, An Improved Boosting Scheme for Regression. *Neural computation* 18, 7 (2006), 1678–1710.
[55] Guangyu Sun, Christopher Hughes, Changkyu Kim, Jishen Zhao, Cong Xu, Yuan Xie, and Yen-Kuang Chen. 2011. Moguls: A Model to Explore the Memory Hierarchy for Bandwidth Improvements. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. IEEE, 377–388.
[56] Teruo Tanimoto, Takatsugu Ono, Koji Inoue, and Hiroshi Sasaki. 2017. Enhanced Dependence Graph Model for Critical Path Analysis on Modern Out-of-order Processors. *IEEE Computer Architecture Letters (CAL)* 16, 2 (2017), 111–114.
[57] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing Data Using t-SNE. *Journal of Machine Learning Research (JMLR)* 9, 11 (2008).
[58] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2020), 4–24.
[59] Ahmad Yasin. 2014. A Top-down Method for Performance Analysis and Counters Architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 35–44.
[60] Joshua J Yi, David J Lilja, and Douglas M Hawkins. 2003. A Statistically Rigorous Approach for Improving Simulation Methodology. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 281–291.
[61] Kai Yu, Jinbo Bi, and Volker Tresp. 2006. Active Learning Via Transductive Experimental Design. In *International Conference on Machine Learning (ICML)*. 1081–1088.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains ArchExplorer's codes and its setup and running descriptions. We provide instructions and click-to-run scripts for reproducing the main results in our paper. Specifically, we reproduce the results of Figure 2, Figure 3, Figure 12, Figure 13, Figure 14, and Figure 15.

A.2 Artifact check-list (meta-information)

- **Code base:** The code base of the artifact involves the entire implementation of ArchExplorer. For example, the artifact contains the new DEG modeling, critical path construction, bottleneck-removal-driven DSE, and automated simulator parallel compilation and simulations.
- **Run-time environment:** We provided a Docker run-time environment for users, which removes large burdens in setting up the environment.
- **Output:** The outputs of the artifact are figures in `PDF` format to reproduce the main results in our paper.
- **Compilation, simulation & modeling:** The artifact includes the automated compilation of GEM5 simulators [9, 39] and the incorporation of power and area modeling using McPAT [38].
- **Benchmarking scripts:** We provide click-to-run benchmarking scripts to evaluate ArchExplorer's performance. Regarding users' different time budgets to run the artifact, we provide three modes to reproduce results. The script `exp_full_mode.sh` can reproduce the "full" results in our paper (we term this reproduction as the full mode), but it costs high runtime and machine resources. The script `exp_partial_mode.sh` demonstrates experimental results with partial benchmarks in a relatively more efficient way compared to the full mode. So, we denote it as the partial mode. The script `exp_demo_mode.sh` is also provided for users to fast experience the functions provided by our artifacts (demo mode). We leverage RISC-V bare model benchmarks [1] in the demo mode.
- **Documents:** We provide detailed `README.md` documents to guide the Docker environment setup, evaluation steps, and results demonstrations.
- **How much disk space required (approximately)?:** The disk space should be larger than 2 TB.
- **How much time is needed to prepare workflow (approximately)?:** It takes several minutes to prepare the workflow if users have SPEC benchmarks [2, 3] and corresponding Simpoint checkpoints [47]. The preparation includes the SPEC benchmarks set up and configurations of some benchmarks. Due to the SPEC license restrictions, we are unable to publicly distribute the SPEC benchmarks. For SPEC CPU2017, users are also required to prepare Simpoints checkpoints. The necessary directory trees for both benchmarks are listed in the `README.md` file, along with detailed instructions on how to map users' provided benchmarks to the pulled Docker environment that we have prepared. In the Docker environment, certain benchmarks need to be reconfigured to support the simulations. Otherwise, the simulation would get stuck and restrict to reproduction of results. The methods to reconfigure these benchmarks are also listed in `README.md`. However, we provide scripts that allow the reproduction of results without reconfiguration of certain benchmarks, *i.e.*, partial mode. As a result, the expected outcomes and results may differ between the partial mode and full mode. However, the generated figures using the partial mode do not affect the conclusions claimed in our paper. Furthermore, results could be different if distinct Simpoints checkpoints are leveraged in the experiments. The Simpoints checkpoints settings are also mentioned in `README.md`.
- **How much time is needed to complete experiments (approximately)?:** For high-end Linux machines, such as 80 cores of Intel(R) Xeon(R) CPU E7-4820 v3 @ 1.90GHz with 1 TB of main memory, the full mode takes approximately 15 days. The partial mode costs about 9 days. The demo mode consumes around 5 hours, but it only reproduces Figure 2 and Figure 3.

A.3 Description

A.3.1 How to access. The artifact is archived in Zenodo [8].

A.3.2 Hardware dependencies. The artifact requires a high-end Linux machine with at least 2 TB of disk space. The main memory should be at least 64 GB to support parallel compilation and simulations.

For reference, we list our system configurations here:

- OS: Ubuntu 18.04
- CPU: Intel Xeon Platinum 8163 CPU @ 2.50GHz (96 cores)
- DRAM: 400 GB
- 9.6 TB

[8] https://doi.org/10.5281/zenodo.8353864

A.3.3 Software dependencies. The software dependencies are resolved by our provided Docker environment. Users are required to support Docker commands in the machines if using our provided Docker environment.

A.4 Installation

The installation requires two steps, as listed below.

A.4.1 SPEC CPU benchmarks installation. This step may take some time since it is necessary to prepare for the workflow. Due to the SPEC CPU benchmarks license, we cannot release benchmarks to the public. Users need to prepare SPEC CPU benchmarks and install them in a manner *w.r.t.* our accepted directories trees. More detailed information about Simpoints settings and accepted directories trees are instructed in the `README.md` file. Assume the SPEC CPU benchmarks have been installed in `/path/to/benchmarks`.

A.4.2 Code installation. Download and decompress the artifact, and pull and install the Docker image.

```
$ unzip arch-explorer.zip
$ cd arch-explorer-main
$ docker run -it -d \
--name micro23 \
--hostname micro23 \
--network=host \
-v $(pwd):/root/workspace/arch-explorer \
-v /path/to/benchmarks: \
  /root/workspace/benchmarks \
-w /root/workspace \
docker.io/troore/arch-explorer:2.0
$ docker exec -it micro23 /bin/bash
```

Since users have already set the directories mapping strategy by executing the Docker "run" command, codes should be placed in `/root/workspace/arch-explorer`, and SPEC CPU benchmarks should be placed in `/root/workspace/benchmarks` when users enter the Docker environment using the "exec" command.

A.5 Experiment workflow

A.5.1 Basic Setup. After users enter the Docker image using the `exec` command, execute the following commands to build configurable infrastructures that ArchExplorer depends on.

```
$ cd /path/to/arch-explorer
$ ./tools/settings.sh
$ export PYTHONPATH=`pwd`
```

A.5.2 Run experiments with three modes. There are three possible factors that will probably prevent users from reproducing our paper results:

- benchmark availability
- hard-coded workload absolute paths
- long runtime

We assume that SPEC CPU benchmarks are available for users. However, if it is not true, we provide a demo mode for users to successfully run through some of our experiments. Under the demo mode, we use open-source RISC-V bare model benchmarks [1] rather than SPEC CPU benchmarks.

Moreover, some SPEC benchmarks MUST contain hard-coded workloads absolute paths, *e.g.*, 464.h264ref from SPEC CPU2006 benchmark. These hard-coded absolute paths would prevent users from reproducing results in a push-button way. Human efforts are required to configure the hard-coded absolute paths before results related to these benchmarks are expected.

Last, the whole process for reproducing all results in our paper will take approximately 15 days on our testing systems (for high-end Linux server machines, *e.g.*, 96 cores of Intel Xeon Platinum 8163 CPU @ 2.50GHz with 400 GB main memory).

Therefore, if users have SPEC CPU benchmarks, and do not want to manually resolve the hard-coded absolute paths, or cannot accept the long runtime, we set the partial mode in which only the benchmarks without hard-coded absolute paths will be run.

We also provide the full mode which can reproduce the experimental results with all utilized benchmarks as in our paper.

Which mode to choose depends on your time budget.

A.5.3 Commands for three modes. In the Docker environment, enter the `artifacts` directory.

```
$ cd arch-explorer/artifacts/
```

The steps for running experiments with three modes are shown below.

```
$ ./exp_demo_mode.sh # demo mode: about 5 \
  # hours, but only reproduce \
  # Figure 2 and Figure 3.
$ ./exp_partial_mode.sh # partial mode: about \
  # 9 days, and can reproduce all figures.
$ ./exp_full_mode.sh # full mode: about \
  # 15 days, and can reproduce all figures.
```

A.6 Evaluation and expected results

Results are figures in `PDF` format. The demo and partial mode have some distinctions from the figures in the paper due to the different workloads utilized in the experiments. However, the generated figures do not affect the conclusions claimed in our paper. Results are stored in respective sub-directories in `artifacts`, and the paths of these results are demonstrated in the `README.md` file.

A.7 Notes

A.7.1 Benchmark reconfiguration for full mode. Before running in full mode, users are required to reconfigure SPEC CPU benchmarks. Otherwise, those benchmarks would fail in simulations and prevent the entire DSE process from continuing to run. These benchmarks are 464.h264ref, 600.perlbench_s, 623.xalancbmk_s, 625.x264_s, and 638.imagick_s, where 464.h264ref is from SPEC CPU2006 benchmarks, and others are from SPEC CPU2017 benchmarks.

For the detailed steps of the reconfiguration, please refer it to in the `README.md`.

A.7.2 About `README.md`. The `README.md` documents of the artifact provide additional information on the illustration of code organizations and detailed steps regarding running experiments.